Models and Analysis in Distributed Systems
Models and Analysis in Distributed Systems

Edited by
Serge Haddad
Fabrice Kordon
Laurent Pautet
Laure Petrucci
Contents

Foreword ........................................ 9
Claude GIRault

Chapter 1. Introduction ........................................ 17
Serge HADDAD, Fabrice KORDON, Laurent PAUTET and Laure PETRUCCI

FIRST PART. FORMAL MODELS FOR DISTRIBUTED SYSTEMS .......... 21

Chapter 2. Introduction to Formal Models .......................... 23
Laure PETRUCCI
  2.1. Motivation ........................................ 23
  2.2. Semi-formal models ................................ 24
  2.3. Formal models .................................... 27
  2.4. After specification, verification ....................... 35
  2.5. Outline of Part I .................................. 37
  2.6. Bibliography ...................................... 37

Chapter 3. Specification and Design Approaches .................... 41
Christine CHOPPY and Laure PETRUCCI
  3.1. Introduction ....................................... 41
  3.2. Criteria for developing specifications ................. 42
  3.3. Specification development methodologies .............. 50
  3.4. Conclusion ........................................ 60
  3.5. Bibliography ..................................... 60

Chapter 4. Modeling Time ......................................... 63
Béatrice BÉRARD
  4.1. Introduction ....................................... 63
  4.2. Semantics of timed models .......................... 65
Foreword

Verification and hence modeling are a mandatory but intricate problem for engineers developing embedded distributed real-time systems that are entrusted with critical safety applications like medical care, transportation, energy production, industrial processes, military operations. Therefore, while emerging 40 years ago, first for circuit design, avionics and finally for all domains, verification environments are now widely exploited by industry and fully integrated into the development processes.

Volume 1 presented design and algorithms for developing these large-scale distributed systems, real-time embedded ones, security concepts for peer-to-peer and ubiquitous computing. However the crucial problem of their correctness is made hard by their complexity, the difficulty of managing fault tolerance, the real-time constraints that they must satisfy, asynchronism of worldly spread units as well as the heterogeneity of devices, networks and applications.

This second volume presents different approaches for mastering these verification problems, beginning with the main concepts and formal methods used for modeling and well structuring distributed systems and for expressing their logical and timed properties. Then it explores the theoretical approaches, mainly logic and automata theory, for behavioral verification of these models. It goes thoroughly into the decidability issues and algorithmic complexity that are the inherent obstacles to overcome particularly for dealing with infinite state spaces and timed models. Collecting the experience of researchers from several laboratories, this volume covers advanced topics of distributed system modeling and verification. It aims at giving a deep knowledge of theory, methods and algorithms to Masters and PhD students as well as to engineers who are already good experts in verification.

Semi-formal specifications and models are a first step for a precise system description. The Unified Modeling Language (UML), widely used in industry, provides diagrams for describing the relations between classes, objects, operations, activities, and allows for examining the system action sequences, reachable states and desired
properties. Such specifications provide a good understanding of the system and allow early detection of some errors. Furthermore, formal models, such as algebraic specification, automata, Petri nets (PN), process algebras, bring abstraction, precision and rigor by precisely describing all the possible behaviors of a system. They allow for performing exhaustive simulation and therefore checking some safety and liveness properties. Moreover temporal logics like Linear Time Logic (LTL) and Computation Tree Logic (CTL) are introduced to express properties of sequences of states generated by these formal models. However the size of the generated sets of states may be so huge that it raises the problems of complexity of the space and time needed to build and explore them. These sets may even be infinite and their manipulation requires sophisticated methods.

Whatever the chosen formalism, system modeling has to keep verification in mind. The abstraction level needs to identify the system elements that must be taken into account, while neglecting those which are out of the verification purposes. Once identified, the system relevant elements are progressively taken into account, and refined. Incremental construction allows us to validate successive refinements. Model oriented engineering approaches may be based on problem frames, architectural or component architectures. Property oriented approaches may use languages like the Common Algebraic Specification Language (CASL) extended with a Labelled Transition Logic to express conditions satisfied by states, requirements on the transitions and incompatible elementary actions.

As modern distributed systems and consequently their models become very large and complex it is important to express their structure. Architecture and Analysis Description Languages (AADL) help to manage and improve it by component composition via interfaces and packages, providing a convenient analysis support in case of future reconfigurations or evolutions.

System verification depends heavily upon the interrelated choices concerning the expressiveness of the formal model, the system requirements and expected properties, the adequate verification methods and moreover the quality of the available tools. Axiomatic proof of properties is highly desirable, but even if computer-aided it needs intensive work for system formalization (much more difficult than modeling) and rigorous checking by highly trained engineers. Moreover, repetitions for each design correction increase cost and delay. Therefore engineers mainly use automatic verification based on numerous model checking methods. Researches combine the advantages and drawbacks of methods, extend the models and develop new strategies. Many subtle variants of models and classes of properties may drastically change complexity in time or in space, or require ingenious extensions of methods and algorithms at the decidability borders. Often expressiveness is costly for analysis. Inhibitor or reset arcs of PN make reachability undecidable. Decidability of counter automata may be obtained by restricting their counters and avoiding zero tests. Association of time with tokens instead of transitions requires more complex constructions for reachability
foreword 11

Proofs. Fortunately some powerful extensions have been defined without theoretical loss. Colored PN still allow positive linear flows and easily understandable invariants, symmetries and parameterization. Recursivity is smartly introduced for PN keeping their most useful properties. Colored semantics of PN by unfolding although rather cumbersome, allows for efficient verification and reductions. PN box calculus allows CCS-like process algebra but nevertheless decidability of verification.

Expression of properties is the most sensitive choice. Generic ones like boundedness, liveness, even home states are useful but not sufficient for verifying the complex behavior of distributed systems. Therefore temporal logics expressing the intricate succession of events are so essential that for the past 40 years much research has focused on them, thus leading to effective automatic verification of complex systems. For this reason, the pioneer fundamental work of E. Clarke, A. Emerson and J. Sifakis has been recognized by the 2007 ACM Turing Award.

The state graph is the key object for verification. Even when finite, it may be exponentially huge w.r.t. the model size so that its reduction is a major goal. Some model simplifications, like agglomerations of actions, allow us to suppress many intermediate states and meaningless interleaving effects, provided suitable conditions preserve behavioral properties. For Petri nets, intricate behavioral conditions on firing sequences provide powerful agglomerations but the help of structural properties simplifies the checking. Avoiding the state graph, structural verifications of PN use flows, invariants, systems of integer linear inequalities and distinguished place subsets. For better behavioral verification, a system is abstracted as an automaton accepting transition sequences of infinite length. To be checked, a Linear-time Temporal Logic formula $\Phi$ is automatically translated into a second automaton, called a Büchi automaton, whose language is the sets of words that contradict $\Phi$. Their “synchronized product” is built to check if they have no common word (emptiness test) and $\Phi$ holds, otherwise a counter example is found.

The memory space for these automata may be reduced by representing only the index states in a hash table, or by neglecting parts of the state space irrelevant for checking a particular formula. Better still, the widely used Binary Decision Diagrams (BDD) provide an extremely compact representation of a state space as a Directed Acyclic Graph of Boolean functions sharing most of their common subexpressions. BDD may grow linearly even when state graphs grow exponentially. They are also extended to represent whole graphs and automata allowing us to check paths and to achieve model checking for CTL as well as for LTL. Variants again provide more compactness (Zero suppressed BDD) or larger scopes (Algebraic DD, Multi-valued DD and Data DD). Interleaving partial executions of actions is a major cause of space explosion; therefore important gains are obtained by using equivalence classes of independent action subsequences. Covering Step Graphs, partial order methods and trace unfoldings lead to many improvements like persistent sets, sleep sets, stubborn sets. Distributed systems often have identical components, modeled by Colored Petri...
Nets (CPN) so their behavioral symmetries allow us to use quotient state graphs and compact symbolic representations. All these improvements now help model checking of large distributed systems, mainly hardware and embedded ones.

Verification of infinite state systems is a challenging problem because all systems use integer and real variables, dynamic data structures, recursive calls, list processing, process creation, parameterization that lead to infinity or unknown bounds. Infinity raises specific difficulties for verification because it requires finite representations with special decision techniques. Only subproblems with strong restrictions become decidable.

Counter systems are finite-state automata with counters, that are non-negative integer variables. Their transition relation is expressed by Presburger formulae which control guards and counter modifications. The Presburger arithmetic allows addition but not multiplication of variables with relational and Boolean connectors and quantifiers. It is one of the most expressive fragments of arithmetic that is decidable. Vector Addition Systems are restricted, hence called “succinct” counter automata without zero test, shown equivalent to Petri nets, and allowing us to decide boundedness and reachability of a target marking. Conversely satisfiability for Presburger arithmetic may be solved by the non-emptiness test for finite state automata. Many tools have been implemented for Presburger decision procedures and for verification of infinite state systems with counters.

Petri nets may be extended by recursion while still allowing for decidability and model checking. Recursive PN (RPN) introduce abstract transitions for creating and resuming processes. The simpler model of Sequential RPN (SRPN) allows us to run only the child process by stacking its father. Each abstract transition is provided with the initial marking of the child process and a Presburger condition for its termination. An “extended marking” is the stack of saved occurrences of abstract transitions having created a child process (modeling the saved interrupt points) with their corresponding marking at the creation (modeling the saved contexts). A termination pops up the stack and fires the abstract transition with the saved marking. SRPN are a fundamental model of great practical and theoretical interest because they are a strict extension of PN able to naturally describe key system features like interruptions, exceptions, management and even fault tolerance while reachability and verification of LTL formulae remain decidable and extended structural linear invariants may be computed.

Real-time systems deal with time, dates and delays which must be carefully taken into account for sound verification and for performance evaluation. The underlying problems are difficult because concurrency and distribution involve subtleties and moreover because continuous time variables require managing dense sets. Markings are extended to also represent clocks or time variables. Different semantics are used for time modeling: event dates and action delays with either clock ticks for discrete time or real variables for continuous time. Timed automata are the basic model
equipped with a finite set of real synchronous clocks. These clocks are synchronously increased by special delay transitions, they may be compared with constants for guards and also be reset by firings. Although they can generate an infinite state space, most decision problems remain decidable. Among Time Transition Systems (TTS) variants, Time PN (TPN) associate time with transitions and model urgency or time out, but they cannot disable transitions that become obsolete. Conversely, Timed PN (TdPN) more subtly associate age to tokens. Their input arcs specify an age interval during which tokens may be consumed, and their output arcs give initial ages to created tokens. Their lazy semantics do not model urgency but allow for disabling transitions when time elapses. The state descriptions and firings become far more complex for all TTS with elapsing of time, durations of transitions, minimum and maximum delays for firing. The often assumed instantaneity of some actions or guard evaluations may be not always compatible with real hardware speed. Temporal logic CTL must be extended for timing, giving in particular CTLT. Strong or weak equivalences between time automata may also be defined. Despite the difficulties, these new logics with their rather efficient and scalable model checking algorithms, provide time modeling and verification for industrial systems.

Real-time systems involve time represented by clocks that are continuous variables forcing us to deal with infinity by means of new “finite-state abstractions”. For the classical timed automata, “configurations” are defined as the product of a state and the set of clock values. These clocks are increased by the delay transitions. Subspaces of configurations are delimited by guards which combine comparisons of clocks with constants (that may be all chosen as integers). An equivalence between configurations allows us to divide this space into regions delimited by the constraints and also by the steps of integer values crossed by the clocks, up to the maximum constant to which the clocks are compared. Some infinite region gathers all the values above this maximum. This partition distinguishes regions from all their boundaries of decreasing dimension, down to elementary points. Thus, the continuous transition system is reduced to a region automaton with a finite number of regions for which reachability and language emptiness are decidable. Finite-state abstraction has been extended to more expressive time automata and to Time Petri nets (TPN). The Büchi automata approach and the emptiness test may be applied to these nets for temporal logic verification of LTL and TCTL formulae.

Problems are far more complex for models with an infinite number of regions. This is the case for Timed Petri Nets (TdPN) because tokens have ages. A more complicated construction makes the coverability problem still decidable by using a finite recurrence to compute the predecessors of the configuration to be covered. The problem of token liveness which arises because some tokens may become useless (i.e. dead) when time elapses, is also shown decidable. A Zeno execution is one where an infinite number of actions or message transmissions must be performed within a finite delay that is unrealistic for real systems. Existence or non-existence of such a sequence is also decidable. Verification of real-time systems being theoretically well
founded, tools have been developed for them. However they raise several axiomatic
issues about clock precision, time progress and Zenoness. Therefore a new semantics
“Almost As Soon As Possible” (AASAP) has been introduced as an emerging research
domain.

Controlling a system G consists in designing an interacting controller C so that the
controlled system (C/G) satisfies its requirements whatever the actions of the environ-
ment of S. However some system actions may be uncontrollable. Also the controller
may only have a partial observation of the state of G. The formal control problems are
very representative of the ones appearing in real peer-to-peer systems and take into
account the complete underlying architecture. Verifying that the controller effectively
keeps the systems within the requirements is simpler than its synthesis which aims at
automatically producing a correctly controlled system.

The completed tasks are represented by the action sequences whose end states are
in the set of terminal states of G, called markers. The controller is modeled by an au-
tomaton S over the same actions and a control function $\Psi$ that indicates whether each
action s of G is enabled by C. By convention it must enable all uncontrolled actions.
L(C/G) is the language of sequences generated by G whose actions are enabled by C.
A subset K of L(G) specifies all the “legal” sequences. K is controllable if any prefix
followed by an uncontrollable action remains legal. If K is not controllable, a fixed-
point algorithm builds a supremal sublanguage of K that is controllable. Local and
modular specifications and solutions have been proposed for decentralized control.
Each controller $C_i$ makes local observations of the system and can disable only a sub-
set of controllable actions. It takes local decisions. An arbiter mechanism is required
for a final decision when there is no consensus between the controllers. Moreover
deadlocks must be avoided. Intricate conditions define tolerance and co-observability
of controllers and distinguish disjunctive or conjunctive arbiters. A general controller
may be obtained by a shuffle of local controllers of the two types. Cooperative con-
trol allows local controllers to exchange information when their local views are not
sufficient.

The synthesis problem for distributed systems is, in some sense, very general. The
system architecture defines a set of processes and their possible communications, A
variable belongs to the partition corresponding to a process if it can be modified by
this process. Indeed at most one process can have a writing access to a given variable.
This very general architecture may be refined. The control is very difficult to achieve:
the problem is in general undecidable in particular for LTL or CTL specifications. For
the decidable sub-cases, the theoretical complexity is very high. For decidability with
LTL specifications, the processes must be organized in a pipeline with restricted access
to variables. For asynchronous communications via shared variables and controllers
with causal memories, the synthesis needs extremely restrictive conditions.
The control problem may be viewed as a game where the controller plays against the environment. A distributed controller is a team of players able to play actions simultaneously but each one does not know the other’s choices. At each state, each player can concurrently choose one move among available ones. Each state is evaluated w.r.t. a set of desired properties. A player strategy is a function determining his moves after a state sequence. The logic ATL (Alternating-time Temporal Logic) is suited to open systems with multiple agents. It can express that some agent can enforce a property. It appears as an extension of CTL offering besides the connectors X, G and U, a selective quantification over paths. For any subset P of players, possibly empty, called a coalition, this new quantifier “P” selects the paths enforced by the strategies of the P players: “P” $\Phi$ means that P can enforce $\Phi$ and the dual $[[P]] \Phi$ means that P cannot avoid $\Phi$. The implemented model checking algorithm extends those of CTL. Like CTL with CTL$^*$, ATL may be extended to ATL$^*$, allowing both state and path formulae. However it cannot express that an infinite number of requests implies an infinite number of grants. Extensions of ATL have been proposed for more expressiveness as well as for games where the players have incomplete information.

This volume provides strong bases for new research extending verification methods in open fields such as composition and refinement, aspect orientation, new algorithms and heuristics, distributed verification, synergies with theorem provers, scheduling, performance evaluation, and more.

Claude Girault, Emeritus Professor
Pierre & Marie Curie University
Chapter 1

Introduction

Problematics

The complexity of dynamic systems grows much faster than our ability to manage them [LEV 97]. In particular, the parallel execution of the threads of a distributed system requires the elaboration of sophisticated models and methods.

The oldest technique, simulation, is a straightforward way to increase confidence about the correctness of an implementation. Such a simulation is based on a model of the system with operational semantics in order to perform the elementary steps of the system. Unfortunately due to the non-determinism of distributed systems, replaying a simulation is a difficult task.

More precisely this problem is a consequence of two factors: the variable transit time of any message and the relative speed of the machine processors. Thus with the help of (vectorial) logical clocks associated with every station, additional information can be managed during the simulation so that it can be replayed [BON 96]. Such mechanisms can easily be integrated within a framework for distributed execution (called middleware).

Even if simulation techniques point out some bugs, they can never fully reassure to the designer of the system. Thus tests must be combined with other techniques in order to obtain a more complete validation of the system.

Introduction written by Serge HADDAD, Fabrice KORDON, Laurent PAUTET and Laure PETRUCCI.
Among these alternative approaches, the more efficient ones are associated with the early stage of the design. The most appropriate method consists of equipping the design step with a formal model such as UML ¹ [CHA 05]. Unfortunately UML is not intended to have an operational semantic and this feature is required for the analysis of distributed systems. In particular, a formal semantic for the component behavior and for their composition is essential in view of checking the properties of the system.

Once a model with a formal semantics is obtained, the properties can be expressed either in a specific way (absence of deadlocks, mutual exclusion, etc.) or in a generic way via some logic (such as temporal logics) that can express fairness, liveness, etc. Furthermore, in order to combine performance evaluation, and verification, quantitative logics have also been introduced.

There are also difficulties with verification methods [LUQ 97]: the competency of the designer, the adaption to industrial case studies, the ability to tackle large-scale applications. Thus pragmatism is a key factor in the success of formal methods: without tools and methodology formal methods would never be adopted by engineers [KOR 03].

**Objective of this book**

The objective of this book is to describe the state of the art in formal methods for distributed and cooperative systems. These systems are characterized by:

– several components with one or more threads, possibly running on different processors;
– asynchronous communications with possible additional assumptions (reliability, order preserving, etc.);
– or local view for every component and no shared data between components.

These are the most common features in modern distributed systems. Numerous issues remain open and are the topic of European research projects. One current research trend consists of intricately mixing the design, modeling, verification, and implementation stages. This prototyping-based approach [KOR 03] is centered around the concept of refinement of models.

This book is more specifically intended for readers who wish to get an overview of the application of formal methods in the design of distributed systems. Master’s and PhD students, and engineers will obtain a thorough understanding of the techniques as well as references for the most up-to-date work in this area.

---

¹ UML stands for Unified Modeling Language.
Organization of the book

This book follows the two stages of a formal method: modeling, and verification.

Part 1 is concerned with the initial step of system design: modeling.
- Chapter 3 discusses a modeling approach to design a consistent specification. After identifying the key elements of the system to be modeled, they are step by step taken into account, and refined until the model is obtained.
- Chapter 4 is devoted to efficient handling of time. Timed models address mechanisms to manipulate time within distributed systems. They make use of discrete clocks and variables, while hybrid systems consider continuous evolution of time.
- Chapter 5 is concerned with the description of software architectures using dedicated languages that are ADLs (architecture description languages).

Part 2 is concerned with the next step of system design: verification.
- Chapter 7 covers the finite-state verification. Historically it is the oldest line of research that has been developed. The main objective of finite-state verification is the reduction of complexity due to the combinatory explosion of the system. Possible approaches are structural methods, which try to avoid developing the behavior of the system, and data representation which reduces the explosion by sharing substructures or exploiting the properties satisfied by the formalism.
- Chapter 8 addresses the problem of verifying infinite-state systems. Most of the research is devoted to the design of formalisms, which are slightly less expressive than Turing machines (or equivalent computational models), and to study which properties are decidable. In this chapter, the main focus is put on extensions of Petri nets and on different variants of counter machines. It emphasizes the fact that small variations lead to drastically different theories.
- Chapter 9 studies timed systems. Time is a particular source of infinity. However, its specificity leads to efficient verification procedures such as those developed for timed automata. Moreover, time can be combined with other sources of infinity such as in time(d) Petri nets. In addition, this chapter tackles the problem of implementing timed systems when the abstractions achieved at the theoretical level (such as the perfect synchronization of the clocks) are no longer satisfied.
- Chapter 10 studies control and synthesis of distributed systems. After recalling the centralized case, it develops, with the aid of specific examples, the specifics of the distributed case and the different possible approaches.
The MeFoSyLoMa community

MeFoSyLoMa (Méthodes Formelles pour les Systèmes Logiciels et Matériels) is an association gathering several world-renowned research teams from various laboratories in the Paris area [MEF 11]. It is composed of people from LIP6 (P. & M. Curie University), LIPN (University of Paris 13), LSV (École Normale Supérieure de Cachan), LTCI (Telecom ParisTech), CÉDRIC (CNAM), IBISC (University of Évry-Val-d’Essonne), and LACL (University of Paris 12). Its members, approximately 80 researchers and PhD students, all have common interest in the construction of distributed systems and promote a software development cycle based on modeling, analysis (formal), and model-based implementation. This community was founded in 2005 and is federated by regular seminars from well-known researchers (inside and outside the community) as well as by common research activities and the organization of events in their domains such as conferences, workshops, or book writing.

The editors of this book, as well as most authors, are from this community.

Bibliographie


---

4. Laboratoire d’Informatique de Paris Nord.
5. Laboratoire de Spécification et de Vérification.
6. Laboratoire Traitement et Communication de l’Information.
7. Centre d’Études et de Recherche en Informatique du CNAM.
8. Informatique, Biologie Intégrative et Systèmes Complexes.
First part

Formal Models for Distributed Systems
Chapter 2

Introduction to Formal Models

2.1. Motivation

Systems that are developed nowadays are of increasing complexity, and their functioning may have dramatic consequences on their environment, even irreversible ones, be it economical or human, e.g. avionics mission systems [PET 03], healthcare management [JØR 04], etc. [Dep 09].

It is thus of the utmost importance to design secure and safe systems, and their behavior must be checked before they become operational.

Verification of such critical systems is usually carried out using methods and techniques that largely depend on the problem at hand. For example, for hardware systems, such as avionics mission systems, physical test-beds are used. In such a case, a plane model, reproducing the exact system, is used on the ground. Simulation traces are logged and analyzed. Such an approach obviously necessitates a hardware infrastructure, as well as qualified staff, to conduct tests. Setting up this kind of test-bed is also usually very time-consuming. Moreover, the simulation traces obtained present the functioning in so much detail that their interpretation is a difficult task. It is thus clear that for verification to be efficient, an adequate abstraction level corresponding to the problem at hand is necessary.

To achieve these safety goals, one should abstract away from the physical system, using a model. Such an approach has many advantages:

– as no hardware is involved, its cost is relatively small;

Chapter written by Laure PETRUCCI.
– it can be analyzed using computer tools, and modified without a significant additional cost;
– the designer of the system model has a better and more rigorous understanding of the characteristics that need to be put into operation;
– once the verification of the expected properties of the model is satisfactory, an experimental prototype can be developed, with sufficient confidence, as many of the design errors have already been eliminated;
– moreover, such a formal specification helps with future maintenance, especially by a person not involved in the initial project.

Several types of specification models and languages can be considered, with complementary goals. They are detailed in the following sections.

2.2. Semi-formal models

System specification can be more or less formal, according to the techniques employed. The use of semi-formal models, such as UML (unified modeling language, [PIL 05]) may achieve part of the intended goals:
– while writing the specification, the designer improves his understanding of the expected behavior of the system, as well as the interactions between its various components. Writing the model enables reconsideration of some conceptual choices, and provides a more accurate understanding of the system itself;
– a model written in a simple notation facilitates communication with clients;
– during a maintenance phase, the model constitutes precise documentation of the system.

A UML specification is composed via various development steps, each of them represented by diagrams, which have the advantage of being relatively easy to understand. Here, we present some of the main aspects of UML.

Firstly, a use case diagram depicts the context: it describes the relationship between use cases and the actors within the system under study. Use cases summarize action sequences carried out by the system. The actors are the external parts (either people or other systems) that interact with the system that is being modeled. Use cases may correspond to several execution scenarios.

EXAMPLE.– Let us consider a car insurance problem. Two actors are involved: the client and the insurance agent. The client may declare an accident to the insurance agent. This operation is composed of several elementary actions, such as filling in a declaration, sending it by electronic or surface mail, etc. The insurance agent may decide, according to the case, to refund the client or not. There are thus several possible scenarios. A corresponding use case diagram is presented in Figure 2.1.
These diagrams clarify the system that will be modeled and enables a better understanding of what should or should not happen without getting lost in technical details. They also help to provide, at a later stage, *test sets* for the validation of the system’s behavior.

The model of a system can be structured into classes, represented by a *class diagram*. The different classes contain their own *attributes* and *operations*, and are linked to one another via *relations* of different kinds.

**EXAMPLE.**— Let us consider the class diagram of our accident declaration example, shown in Figure 2.2. A vehicle has several associated attributes, such as its license plate number, its type, and its brand. Similarly, a client has a name, a surname, and an insurance number. The client can execute a `declare()` operation, in order to declare an accident. A declaration concerns exactly one vehicle and one client.

Each object in the system implements part of the intended functionalities. The global behavior is obtained through the cooperation between the different objects. These communications between objects are realized by exchanges of messages, which can be described in a *communication diagram*. 
EXAMPLE.— The communication diagram in Figure 2.3 shows that the client should declare an accident by sending a message declare. He can then send the declaration to the insurance agent who makes a decision in order to reply to the client.

![Communication diagram](image)

Figure 2.3. Communication diagram

This information can also be represented by sequence diagrams, which, in addition, show how objects are created.

EXAMPLE.— The sequence diagram in Figure 2.4 indicates message exchanges between the different objects. Note that the operation initiated by the client to declare an accident also generates a declaration object.

![Sequence diagram](image)

Figure 2.4. Sequence diagram

The flow of information in the system is represented by an activity diagram, which gives additional information complementary to those of the communication and sequence diagrams. It depicts the system evolution as seen by one of its actors.

EXAMPLE.— From the insurance agent point of view, the activities take place as pictured in Figure 2.5: he receives a declaration, handles it, and then sends his reply to the client.

![Activity diagram](image)

Figure 2.5. Activity diagram

A behavioral description of classes, use cases, actors, etc. is described through a state diagram. During the system evolution, the involved entities change state. This may be the result of external events triggering a particular activity.
EXAMPLE. – The state diagram for our accident declaration example is shown in Figure 2.6. The insurance agent initially waits. When he receives a declaration, he moves to a new state in which he is ready to handle it. After checking the declaration, two cases may occur: either there is no problem (OK) and the reimbursement can take place (a positive response is sent to the client), or there is a problem (NOK) and the insurance agent sends a negative response to the client.

Software tools enable a complex specification to be built. They check the consistency among the different diagrams. Moreover, they propose automatic generation of code and test scenarios.

Some diagrams, other than those mentioned in this section, exist, but their presentation is outside the scope of this book. For further information, see e.g. [PIL 05].

To conclude, semi-formal techniques such as UML enable the characteristics of the system to be studied in depth during modeling, with a high level of abstraction, and the implementation detail can be addressed. The diagrams thus obtained provide a good view of the different aspects of the system. However, even though these diagrams are easy to understand for a non-specialist, grasping the full picture can be a challenge. Moreover, system validation cannot be exhaustive. Therefore, formal models aim to tackle these problems.

2.3. Formal models

Formal models enable the correct behavior of a system to be formally (i.e. mathematically) proven. Then, whatever the evolution of the system, it is guaranteed to behave as expected.
In addition to the advantages of the previous techniques, formal models offer analysis tools for the system under study:

– simulation provides evidence of the correct behavior or eventually errors to be discovered, especially severe ones. Correction at this stage is relatively cheap, compared to the correction of an already existing system, either implemented as software or hardware;

– exhaustive verification of system behavior can be performed. First, simulation is applied, leading to a coarse-grained debugging, and then the model is refined by verification of its expected properties.

A wide range of specification models and languages exists. In this book, we shall focus on algebraic specifications, automata [BÉR 01a], Petri nets [GIR 03, DIA 09], and process algebras [BER 01b]. This is justified by the existence of a specification methodology, the possibility of using structural methods, and introducing temporal constraints in the models considered. Finally, architecture description languages (ADLs) [MED 00] enable model composition.